

DESIGN AND IMPLEMENTATION OF AN APPLICATION SPECIFIC INSTRUCTION SET PROCESSOR USING LISATEK DESIGN METHODOLOGY

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF**

**Bachelor of Technology
in
Electronics and Instrumentation Engineering**

**By
Anup Sarma
and
Soubhagya Sutar**



**Department of Electronics and Communication Engineering
National Institute of Technology Rourkela
May 2011**

DESIGN AND IMPLEMENTATION OF AN APPLICATION SPECIFIC INSTRUCTIONSET PROCESSOR USING LISATEK DESIGN METHODOLOGY

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
in
Electronics and Instrumentation Engineering

By
Anup Sarma
and
Soubhagya Sutar

Under the Guidance of
Prof. K K Mahapatra



Department of Electronics and Communication Engineering
National Institute of Technology Rourkela
May 2011



NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

CERTIFICATE

This is to certify that the thesis entitled “Design and Implementation of Application Specific Instruction Set Processor using LISATek Design Methodology” submitted by Anup Sarma (107EI018) and Soubhagya Sutar (107EI010) in partial fulfilment of the requirements for the award of Bachelor of Technology in Electronics and Instrumentation Engineering at National Institute of Technology Rourkela is an authentic work under my supervision and guidance.

To the best of my knowledge, the matter embodied in this thesis has not been submitted to any other University/Institute for the award of any degree or diploma.

Date: 16.05.2011

Place: Rourkela

Prof. K K Mahapatra

Department of ECE

National Institute of Technology Rourkela

Rourkela, Orissa-769008

India

ACKNOWLEDGEMENT

Many people have been involved, directly or indirectly, in the completion of this thesis and we would like to take this opportunity to express our gratitude to them.

We would first like to sincerely thank my project guide, Prof K K Mahapatra, for being a constant source of inspiration throughout the course of this project work and this work would not have been possible without his experienced and valuable guidance.

We would like to express our gratitude to all the research scholars whose works were referred to by us during the completion of this project work. A special mention about Mr. Ayaskanta Swain, Mr. Vijay Sharma and Mr.Jagannath Mohanty for their timely and valuable guidance that helped us to finish the work in the stipulated period of time.

Anup Sarma
107EI018

Soubhagya Sutar
107EI010

ABSTRACT

An Application Specific Instruction Set Processor (ASIP) is widely used as a System on a Chip Component. ASIPs possess an instruction set which is tailored to benefit a specific application. Such specialization allows ASIPs to serve as an intermediate between two dominant processor design styles- ASICs which has high processing abilities at the cost of limited programmability and Programmable solutions such as FPGAs that provide programming flexibility at the cost of less energy efficiency. In this project our goal is to design ASIP, keeping in mind Digital Image and Signal Processing Applications, followed by implementation in hardware (FPGA). The platform used for design is CoWare, which allows processor architecture to be defined at an abstract level and automatic generation of chain of software tools like assembler, linker and simulator for functional verification followed by RTL level description which is to be used for FPGA implementation.

TABLE OF CONTENTS

ABSTRACT	5
LIST OF FIGURES.....	9
LIST OF TABLES	10
Chapter 1	11
INTRODUCTION	11
Chapter 2	14
LISATek DESIGN METHODOLOGY	14
2.1 <i>Advantages of LISATek Design Methodology</i>	16
2.2 <i>Design Flow</i>	16
Chapter 3	17
PROCESSOR DESCRIPTION IN LISA	17
3.1 <i>Hardware Modelling (RESOURCE Section)</i>	18
3.2 <i>Software Modelling</i>	19
3.3 <i>Special operations used</i>	20
Chapter 4	21
PROCESSOR SPECIFICATION	21
4.1 <i>Description</i>	22
Chapter 5	25

INSTRUCTION SET	25
5.1 Abbreviations Used	28
Chapter 6	29
HDL CODE GENERATION	29
Chapter 7	32
SAMPLE APPLICATION	32
7.1 Algorithm	33
Chapter 8	35
WORK FLOW	35
8.1 Description	36
Chapter 9	39
SIMULATION	39
9.1 Simulation in CoWare Processor debugger	40
9.2 Simulation in ModelSim	41
Chapter 10	43
RESULTS	43
Image 1	45
Image 2	46
Chapter 11	49
SYNTHESIS AND PHYSICAL IMPLEMENTATION	49
11.1 RTL Schematic	52

Chapter 12	56
CONCLUSION AND FUTURE WORK	56
REFERENCES	57

LIST OF FIGURES

Fig. 2.1	ASIP Exploration and Implementation Phase.....	16
Fig. 4.1	Pipeline Stages.....	24
Fig. 6.1	Schematic of Generated HDL.....	30
Fig. 7.1	Histogram Equalization.....	33
Fig. 8.1	Work Flow.....	36
Fig. 9.1	Simulation steps in Processor Debugger.....	40
Fig. 9.2	Simulation steps in ModelSim.....	41
Fig. 9.3	Processor Debugger Screenshot.....	42
Fig. 9.4	ModelSim Screenshot.....	42
Fig. 10.1	Original 64x64 Image.....	45
Fig. 10.2	Matlab Processed Image.....	45
Fig. 10.3	Processor Output Image.....	45
Fig. 10.4	Error/Difference Image.....	45
Fig. 10.5	Histogram Comparison of 64x64image.....	45
Fig. 10.6	Original 128x128 Image.....	46
Fig. 10.7	Matlab Processed Image.....	46
Fig. 10.8	Processor Output Image.....	46
Fig. 10.9	Error/Difference Image.....	46
Fig.10.10	Histogram Comparison of 128x128image.....	47
Fig. 11.1	VHDL File structure.....	49
Fig. 11.2	Complete RTL Schematic.....	51
Fig. 11.3	RTL Schematic of Processor and Memory block.....	52
Fig. 11.4	Processor RTL Schematic.....	53
Fig. 11.5	Digital Clock Manager.....	54
Fig. 11.6	ChipScope Pro Screenshot.....	54

LIST OF TABLES

Table 1.1	Comparison between ASIC and Programmable solution.....	12
Table 4.1	Processor Specification.....	22
Table 11.1	Device Utilisation Summary.....	50

Chapter 1

INTRODUCTION

A **digital signal processor (DSP)** is a specialized microprocessor with an optimized architecture for the fast operational needs of digital signal processing. Digital signal processing algorithms typically require a large number of mathematical operations to be performed quickly and repetitively on a set of data. Most general-purpose microprocessors and operating systems can execute DSP algorithms successfully, but are not suitable for use in portable devices such as mobile phones and PDAs because of power supply and space constraints. A specialized digital signal processor, however, will tend to provide a lower-cost solution, with better performance, lower latency, and no requirements for specialized cooling or large batteries.

By the standards of general purpose processors, DSP instruction sets are often highly irregular. One implication for software architecture is that hand optimized assembly is commonly packaged into libraries for re-use.

An **application-specific integrated circuit (ASIC)** is an integrated circuit customized for a particular use, rather than intended for general-purpose use. Modern ASICs often include entire 32-bit processors, memory blocks including ROM, RAM, EEPROM, Flash and other large building blocks. Such an ASIC is often termed a SoC (System on Chip).

	Advantages	Disadvantages
ASIC	High Performance Lower Area Low Power	No Flexibility(Fixed Hardware)
Programmable solution (FPGA, PLA, PLA, etc)	Programmable Reusable Flexible Time to market	Low energy efficiency

Table 1.1 Comparison between ASIC and Programmable solution

The solution is a compromise between ASIC and a programmable solution. ASIP or Application Specific Instruction-Set Processor has architecture close to ASIC with respect to energy efficiency and size and offering limited programmability. These processors are application specific and the instruction set is designed to meet the application needs only. It provides an optimum balance between:

- Computational efficiency (MOPS/mW)
- Reuse opportunities and flexibility
- Cost
- Power consumption

Chapter 2

LISATek DESIGN METHODOLOGY

Traditionally processor design is a long iterative process based on the results of hardware and software simulation. This usually requires a huge team of hardware and software experts who based on these results incorporate change in the architecture and refine the model gradually. The steps involved can be broadly categorized as:

1. Designing an instruction set
2. Design a micro-architecture
 1. Drawing functional blocks
 2. Writing micro-steps
3. Writing a simulator
4. Writing an assembler
5. Implementing the micro-architecture
6. Implementing the control unit
7. Testing and Debugging (the most time consuming task)

This usually involves a huge design time as well as a lot of resources both manpower and cost.

The LISATek product family is an automated embedded processor design and optimization environment which helps to accelerate design of both custom and standard processors. These processors include Application Specific Instruction Set Processors (ASIPs) that are increasingly essential to convergent System on a Chip Functionality. The Design Methodology is based on architecture description language called LISA (Language for Instruction Set Development), which is used to create an abstract model of the processor.

The Language for Instruction Set Architectures is widely used for formal processor description, their peripherals and interfaces. Its orientation is between structural and behavioral description. The language is intended to serve as an intermediate between purely structured language such as VHDL and Instruction set languages. Based on this sole architecture model, subsequent development steps are performed. The tool enables to generate assembler, linker and processor simulator to functionally verify the operation of instruction set and the assembler.

2.1 Advantages of LISATek Design Methodology

- Description of processor architecture from a higher level of abstraction other than RTL level
- Automatic generation of all required software tools – Compiler, Assembler, Linker
- Automatic generation of a RTL model

2.2 Design Flow

Defining a programmable platform tailored to a specific application domain

- Instruction Set definition
- Software Development Tools
- Hardware Implementation : RTL ASIP model

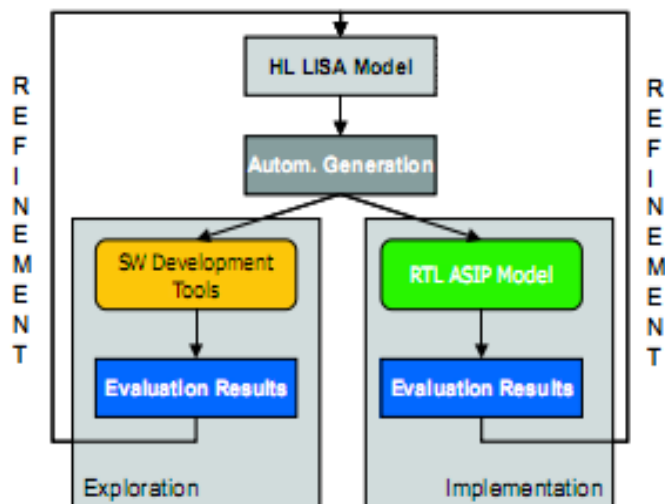


Fig.2.1 ASIP Exploration and Implementation Phases

Chapter 3

PROCESSOR DESCRIPTION IN LISA

A LISA 2.0 processor description mainly consists of hardware and software model of the architecture. The hardware model comprises the definition of processor resources like registers, memories, pipeline and buses for accessing memories. The software model comprises the definition of processor instruction set, their binary instruction coding, assembly syntax and response of the processor hardware to instruction.

3.1 Hardware Modelling (RESOURCE Section)

The RESOURCE section lists the definitions of all objects which are required to build the memory model and the resource model. These resources represent the current state of the processor. Each time the processor performs one control step - this can be an instruction, cycle, phase - the processor is driven into a new state according to the behavior of the functional units.

The Resource declarations follow the style of variable declaration in C and data values of resources are treated like variables in C. However, they can only be declared outside the scope of operations. Consequently, all resources are defined globally and visible for all operations. This resembles the properties of hardware components that are global by their nature. The RESOURCE section allows the declaration of the following types of objects:

Simple Resources:

1. Register and Register Flags
2. Ideal Memory Arrays
3. Signals and flags
4. Other resources that are not visible in the architecture

Memory Maps:

1. The mapping of memories into processor addresses space.
2. The connectivity between memory and bus modules.

The Pipeline structure for instruction and data paths

Pipeline registers storing data on its way from one pipeline stage to the next.

And some no ideal memories such as Caches, Buses etc. as a part of memory subsystem.

3.2 Software Modelling

The software model of the processor consists of processor instructions that are implemented using operations. An operation represents a basic entity in LISA 2.0 model. They represent behavior, structure and instruction set of the programmable architecture. The execution of any instruction ultimately leads to the execution of corresponding instruction. An operation possesses several attributes which are described below.

- The CODING section describes the binary image of the instruction word which is part of the instruction-set model.
- The SYNTAX section describes the assembly syntax of instructions and their operands, which is part of the instruction-set model.
- The BEHAVIOR and EXPRESSION sections describe components of the behavioral model. During simulation, the operation behavior is executed and modifies the values of resources, which drives the system into a new state.
- The ACTIVATION section describes the timing of other operations relative to the current operation.(Used when pipelined model is being used)

Each tool derived from LISA models needs a subset of these sections to perform its particular task. For example, the Assembler utilizes the information contained in the CODING and SYNTAX sections, since this tool maps the instruction syntax onto a binary coding word, while the De-assembler works the other way round. For this reason, it also makes use of the

CODING and SYNTAX sections. The Simulator depends on the information of the BEHAVIOR sections, while the operation scheduler relies on the ACTIVATION section.

3.3 Special operations used

OPERATION decode: This operation represents root of the coding tree. In other words this can also be said to be the instruction decoder. The name of the instruction decoder can be chosen to be anything (here decode), except for two reserved keywords main and reset.

OPERATION main: This operation implements the functionality that has to be executed for a single processor control step. Thus, OPERATION main which is executed at every control step, invokes decode operation which in turn executes the associated instruction.

OPERATION reset: The task of this operation is to bring the processor resources to initial value or initial state.

Chapter 4

PROCESSOR SPECIFICATION

The designed processor is based on a 32-bit architecture with the following specifications:

Instruction Length	32 bits
Opcode Length	8 bits
GPR	24 bits (8 nos)
Program Counter	32 bits
Pipeline Stages	3
Program Memory	32 bits
Range	0x0000-0x0FFF
Data Memory	24 bits
Range	0x0000-0x0FFF

Table 4.1 Processor Specification

4.1 Description

Instruction Length: The total number of bits used to represent an instruction including the opcode and operands. The current instruction being executed is stored in the instruction register (IR).

Opcode Length: The number of bits reserved for the opcode segment of the instruction. Since the opcode length is 8-bits, the total number of instructions that can be defined is $2^8 = 256$.

GPR (General Purpose Register): Eight GPRs each 24 bits in length have been defined to store any transient data required by the program.

Program Counter (PC): The Program Counter keeps track of the next instruction to be executed. It stores the address of the program memory location corresponding to the next instruction. The IR in turn is loaded with the instruction from the memory location pointed to by the PC at the start of the next clock cycle. Since the PC is 32 bits in length, it can point to a program memory with 2^{32} locations.

Program Memory: The program memory stores a sequence of instructions comprising the program that implements the application functionality. The address space of the program memory for the designed architecture is 2^{32} as the PC length is 32-bits. However, due to resource constraints of the final implementation platform, the range is kept limited to 0x1000 memory locations (0x0000-0x0FFF). Also, each memory location is 32-bits long in accordance with the IR length which is also 32-bits long.

Data Memory: The data memory stores the input data on which the program operates as well as the final results computed from the operation. Each memory location is 24-bits in length i.e., same as that of a GPR and hence data can be transferred from memory to GPR in just one clock cycle. The defined range is 0x0000-0x8FFF (0x9000 locations).

Pipeline: A computer program is, in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the functionality of the program. This is known as instruction-level parallelism and implemented through pipelining. The designed architecture implements a 3-stage pipeline. The three stages comprise of the following stages:

- **FD (Fetch and Decode):** Fetching the instruction from the IR and decoding its functionality.
- **AG (Address Generation):** Generating the address of the data memory locations that store the required operands. Also, the branch address in the program memory as specified by the flow control (branch) instructions is generated in this stage.
- **EX (Execution):** Actual execution of the fetched instruction occurs in this stage.

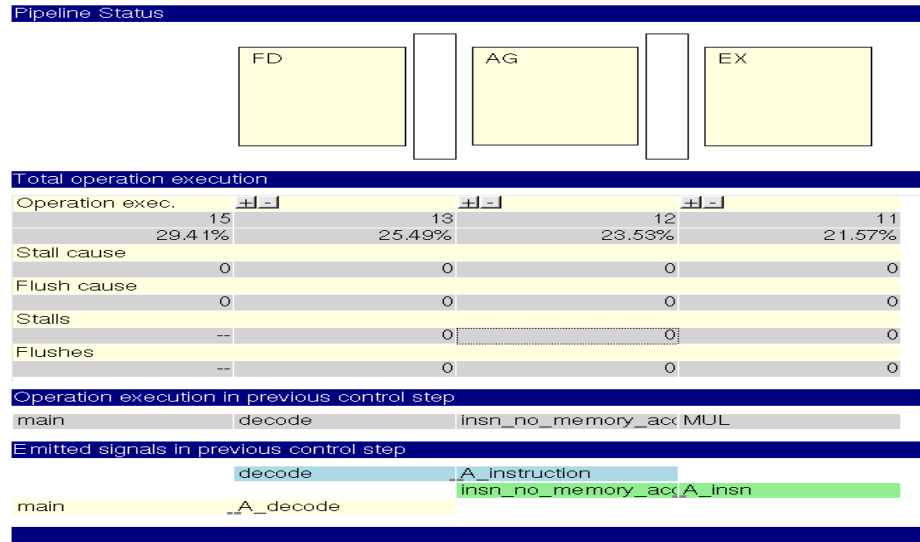


Figure 4.1 Pipeline Stages

FD and EX stages occur for every instruction. AG stages occur only for those instructions that require data memory access or involve branching in program memory (Flow control instructions).

Chapter 5

INSTRUCTION SET

	Syntax	Description
ADD	ADD dst,src1,src2	Adds src1 and src2, stores the result at dst
SUB	SUB dst,src1,src2	Subtracts src2 from src1, stores the result at dst
MUL	MUL dst,src1,src2	Multiplies src1 and src2, stores the result at dst
SHR	SHR dst,src1,src2	Shifts the contents of src1 to the right by the number of bits specified by src2, stores the result at dst
SHL	SHL dst,src1,src2	Shifts the contents of src1 to the left by the number of bits specified by src2, stores result at dst
AND	AND dst,src1,src2	Bitwise ANDs the contents of src1 and src2, stores the result at dst
OR	OR dst,src1,src2	Bitwise ORs the contents of src1 and src2, stores the result at dst
XOR	XOR dst,src1,src2	Bitwise XORs the contents of src1 and src2, stores the result at dst

NOP	NOP	Performs no operation
BZ	BZ @addr	Branch if "Zero"- Program execution transferred to address addr if zero bit is set i.e., result of the last executed arithmetic instruction is zero.
BNZ	BNZ @addr	Branch if "Not Zero"- Program execution transferred to address addr if zero bit is clear i.e., result of the last executed arithmetic instruction is non-zero.
BGEZ	BGEZ @addr	Branch if "Greater Than Equal to Zero"- Program execution transferred to address addr if result of the last executed arithmetic instruction is greater than or equal to zero
BLEZ	BLEZ @addr	Branch if "Less Than Equal to Zero"- Program execution transferred to address addr if result of the last executed arithmetic instruction is less than or equal to zero
B	B @addr	Unconditional branch - Program execution transferred to address addr unconditionally.
RPT	RPT n1,n2	Repeats the next n1 number of lines of the program code n2 number of times.
MOV	MOV src1->dst MOV addr->ar MOV dst->[ar]	Moves the data stored at src1 into dst. Moves address addr into address register ar. Moves data stored at dst into the memory address location specified by address register ar.

5.1 Abbreviations Used

src1, src2: General Purpose Registers (R[0] – R[7]) or immediate operands (e.g. #0x0005)

dst: General Purpose Registers (R[0] – R[7])

addr: Program memory address (0x0000-0xFFFF)

ar: Address Register (A[0]-A[2])

n1, n2: Immediate operands

Chapter 6

HDL CODE GENERATION

The CoWare processor generator is used to create an implementation model of the target architecture, which is a tool present in Processor Designer. The output of the Processor Generator is Hardware Description Language (HDL) code, which can be processed by standard synthesis tools. It can be used to create a VHDL or Verilog processor model of the architecture.

While generating the HDL code the following specifications have been used:

- Configuration for generic memory interface enabling pin true connection to various memories have been selected.
- VHDL is chosen as target backend.
- Asynchronous active low reset type.
- Simulator Script Generation is done corresponding to Mentor Graphics ModelSim.
- Both program and data memories are specified separately as external to the processor.
This allows treating the system as two independent components – the processor and memory part and allows tempering as and when required.

The Generated HDL structure can be viewed as having the following structure.

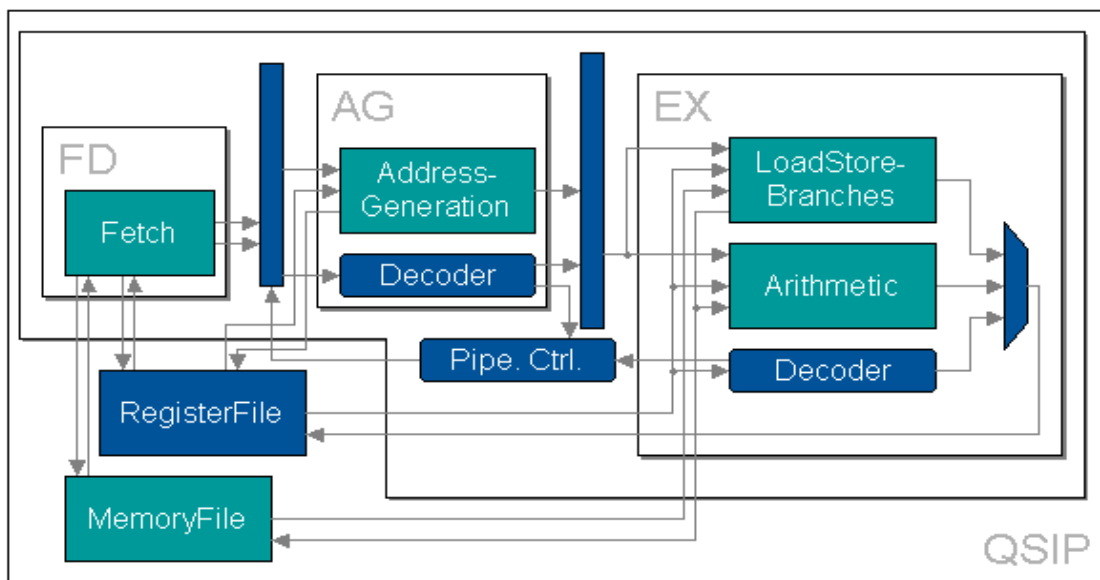


Fig.6.1 Schematic of Generated HDL

As shown in figure, the QSIP structure consists of three stage pipeline elements- FD (fetch/decode), AG (address generation) and EX (execution) stage along with Register File and Memory File. The flow consists of FD part reading the instruction word and propagation of instruction word through the pipeline. Not all instructions require memory accessing. The AG block determines the type of instruction (memory accessing/non accessing) and this information is sent as feedback through the pipeline controller .In the execution stage the final execution takes place and final output values are written back to the designated register or memory locations.

Chapter 7

SAMPLE APPLICATION

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram.

This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.

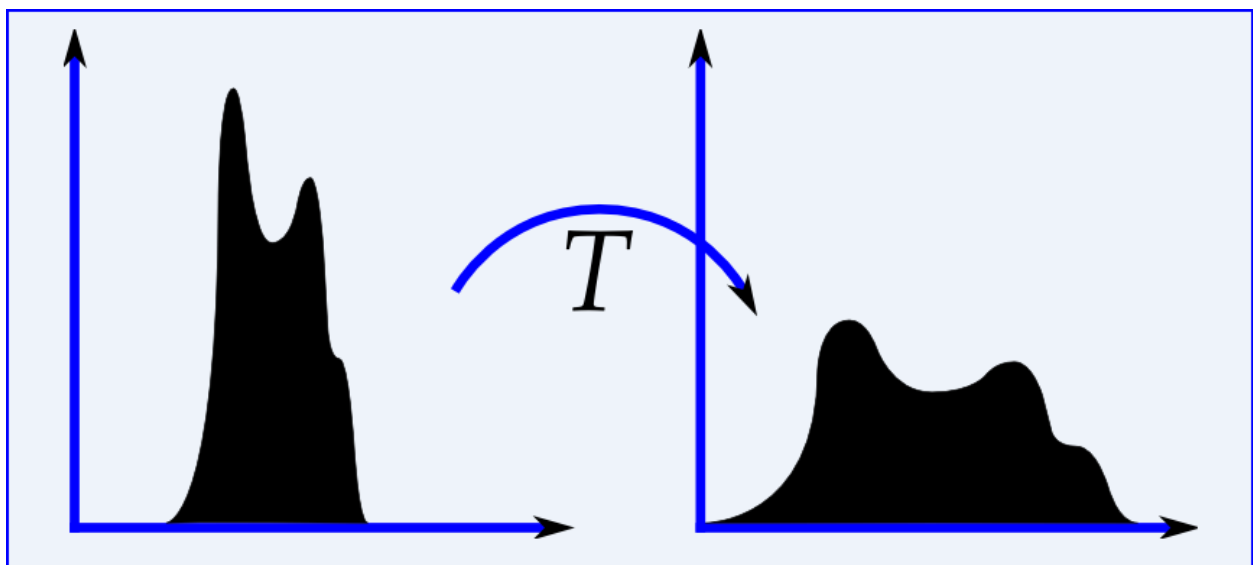


Fig.7.1 Histogram Equalization

(*X-axis*: Pixel Intensity Level, *Y-axis*: Frequency)

7.1 Algorithm

1. Read input image ($M \times N$ pixels) and store corresponding intensity level of each pixel. The intensity levels range from 0-255 (0 – complete black, 255 – complete white). The total number of intensity levels is L ($= 256$) for an 8-bit grayscale image.

2. Count number of pixels (n_k) corresponding to intensity level r_k (k varies from 0-255). $P_r(r_k)$ denotes the probability of occurrence of intensity level r_k in the input image and given by

$$P_r(r_k) = \frac{n_k}{MN} \quad k = 0, 1, 2, \dots, L - 1$$

3. Obtain the replacement value for each intensity level as per the following :

$$\begin{aligned} S_k = T(r_k) &= (L - 1) \sum_{j=0}^k p_r(r_j) \\ &= \frac{L - 1}{MN} \sum_{j=0}^k (n_j) \quad k = 0, 1, 2, \dots, L - 1 \end{aligned}$$

4. Replace the original intensity levels with the corresponding replacement values to generate the histogram equalized image.

In the next chapter, we explore the work flow and implementation of this algorithm in different simulation platforms.

Chapter 8

WORK FLOW

The complete work flow for the image processing application (Histogram Equalization) can be visualized below.

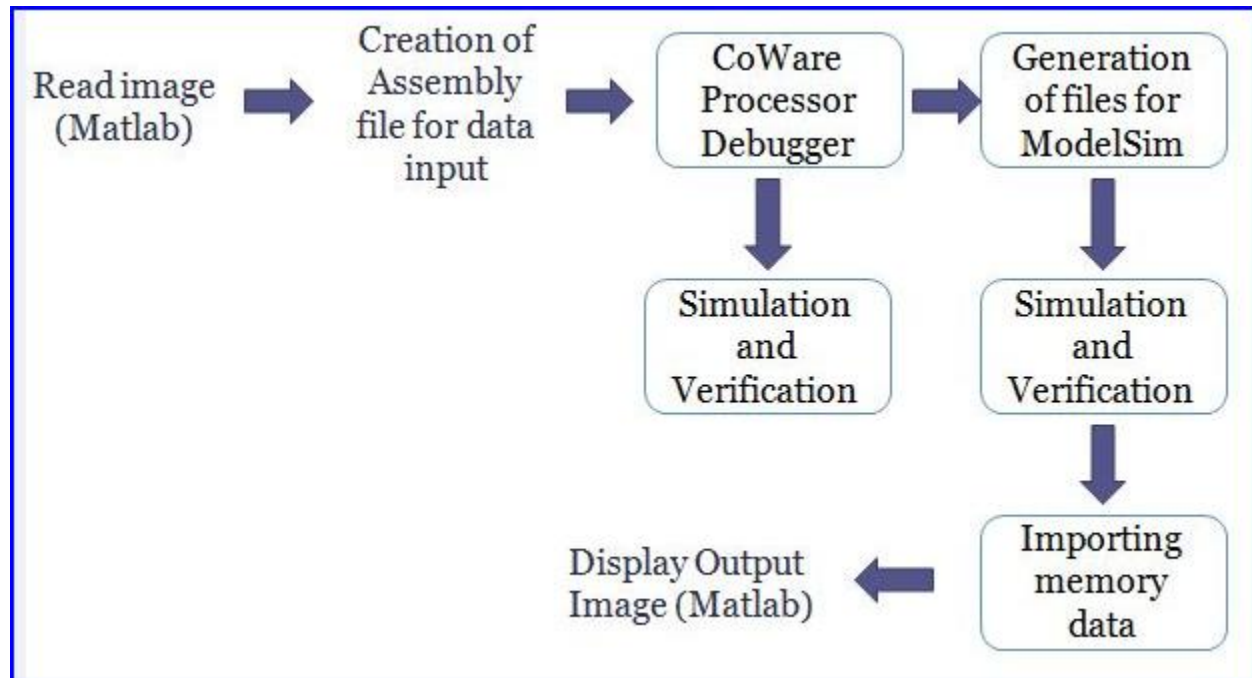


Fig.8.1 Work Flow

8.1 Description

Read image (Matlab): The input image is read using the “imread()” in-built function in Matlab. This stores the intensity levels of each pixel of the image into a matrix specified by a variable.

Creation of Assembly file for data input: Since the number of pixels involved is quite high (‘4096’ for a 64x64 image and ‘16384’ for 128x128 image), we need a easy and fast method to input the intensity values of these pixels into the model processor simulation platform. Of course manual entering of these values during simulation run or final implementation run-time is not possible as it is tiresome and hugely time-consuming. So the data is entered prior to simulation and run-time through an assembly code containing suitable data transfer instructions for loading the data memory with the intensity level values. Even this assembly code is very lengthy and

hence generated using a C program that takes the intensity level values read from the image by Matlab as input and outputs the required assembly code.

CoWare Processor Debugger: The assembly code for the Histogram Equalization application is run on CoWare Processor Debugger. Processor Debugger allows step-wise execution of the instructions and hence the effect of the instruction run on the resources and data memory can be monitored at each step. This allows debugging of the program during the program run and to remove programming errors (if any). The figure below shows a screenshot of the Processor Debugger.

The processed data values that was loaded into the data memory at the end of program execution were verified with Matlab processed values. The 'histeq' in-built function in Matlab carries out histogram equalization of an image. The difference in the two set of values is shown in the form of an error/difference image in "Results" chapter. The histogram of the input image and the histogram equalized (output) image is also shown.

Generation of memory content files for ModelSim: The memory-content files read by the provided simulation memories have a format that can be read by both VHDL and Verilog models. While for simple configurations it is possible to generate the memory-content files by hand, this is not appropriate for more complex applications. To automate the generation of the memory-content files, the Processor Generator generates a memory-layout file, called hdl_memory_configuration.txt. This file can then be processed by a special utility 'exe2txt' (provided by LISATek) to generate the memory-content files together with the executable of an application.

Importing memory data: Once the simulation is run, output values are written and stored into the data memory locations. These values are simply imported as in text format and suitably processed.

Display output image (Matlab): The simulation in ModelSim yielded the processed data values i.e., the new intensity values of the pixels. These values were imported to Matlab and used to display the histogram equalized image. Both the input and the processed output image are shown in “Results” chapter.

Chapter 9

SIMULATION

In the previous chapter, simulation in CoWare Processor Debugger and in ModelSim has been treated as a black box. Here we explain the vital steps involved in the simulation process.

9.1 Simulation in CoWare Processor debugger

The simulation in CoWare processor debugger can be described by following three step process.

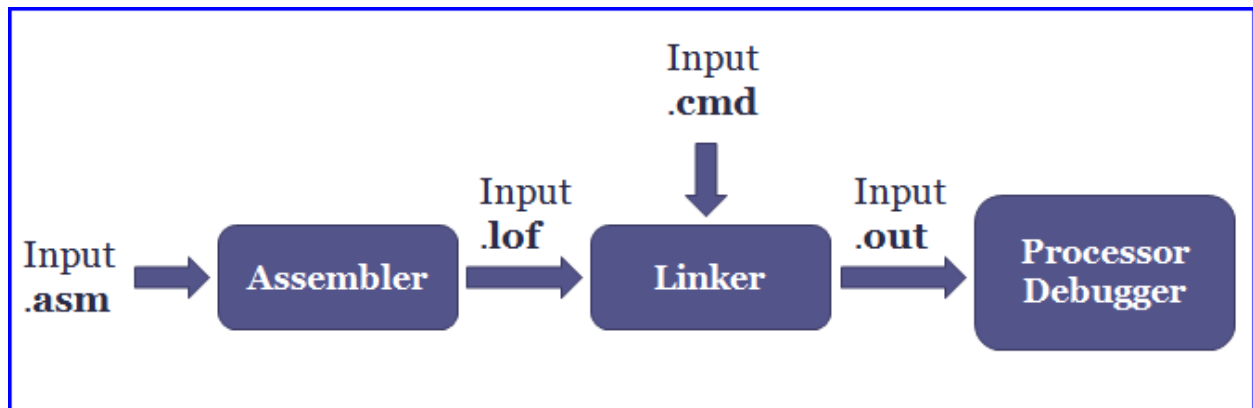


Fig.9.1 Simulation steps in Processor Debugger

The assembler and the linker are the standard software tools generated by the LISATek product family. An assembly file is written corresponding to the instruction set defined for the processor. The assembler takes this file as input and outputs a ‘.lof’ file, among other files. The linker takes in the ‘.lof’ file as input and together with ‘.cmd’ file having specified format, it generates the executable ‘.out’, which can be used to run on virtual QSIP architecture.

9.2 Simulation in ModelSim

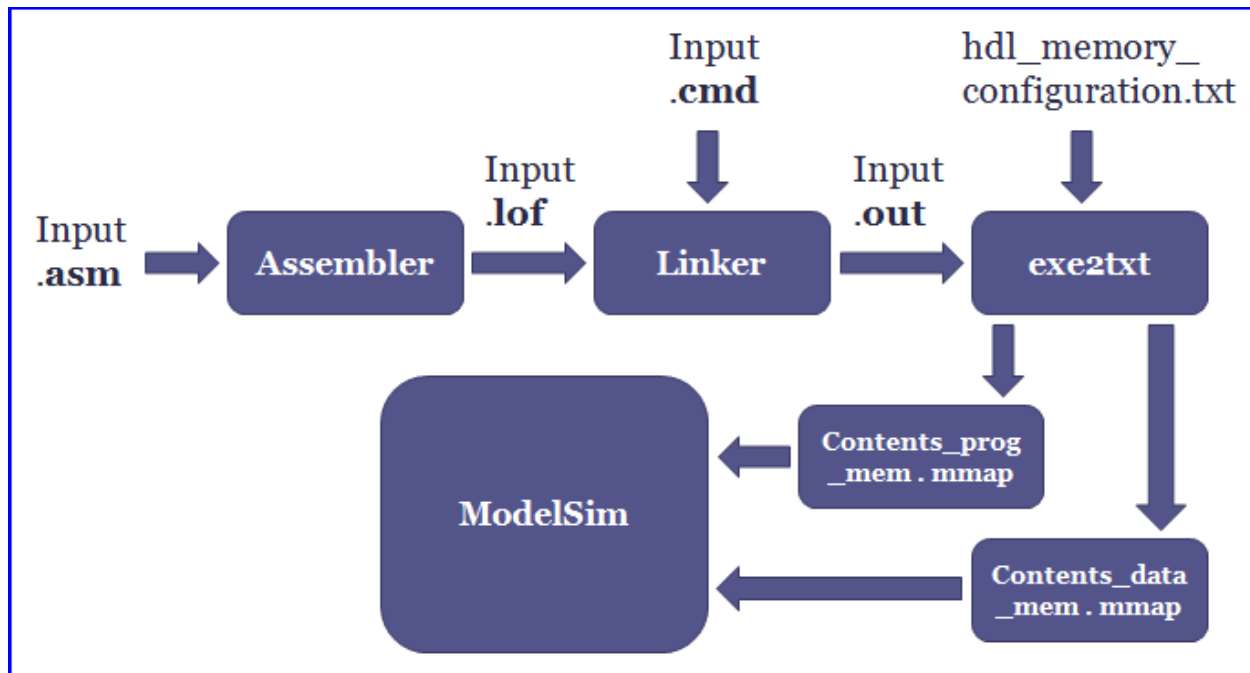


Fig.9.2 Simulation steps in ModelSim

Simulation in ModelSim involves creation of memory definition files using the exe2txt utility provided by CoWare. It takes the executable generated in previous step and the memory configuration file as input and as shown in the figure, the output of the ‘exe2txt’ application is two files, ‘contents_prog_mem.mmap’ and ‘contents_data_mem.mmap’, which is copied into the work directory of the project created by using the VHDL files of the processor model. When simulation is run, as soon as reset signal is asserted the program and data memories read these files (filenames are supplied as parameter) and loaded with values generated at the corresponding address locations.

Screenshots of CoWare Processor Debugger and Mentor Graphics ModelSim during simulation are shown in the next page.

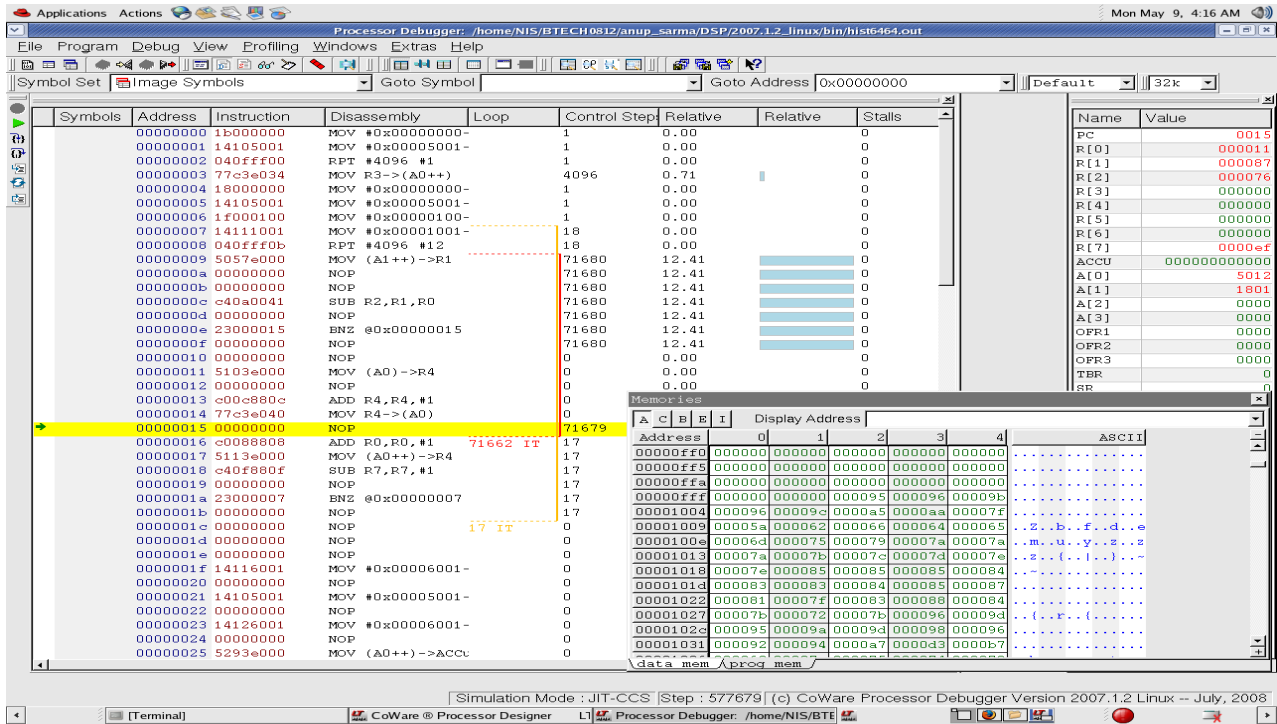


Fig. 9.3 Processor Debugger Screenshot

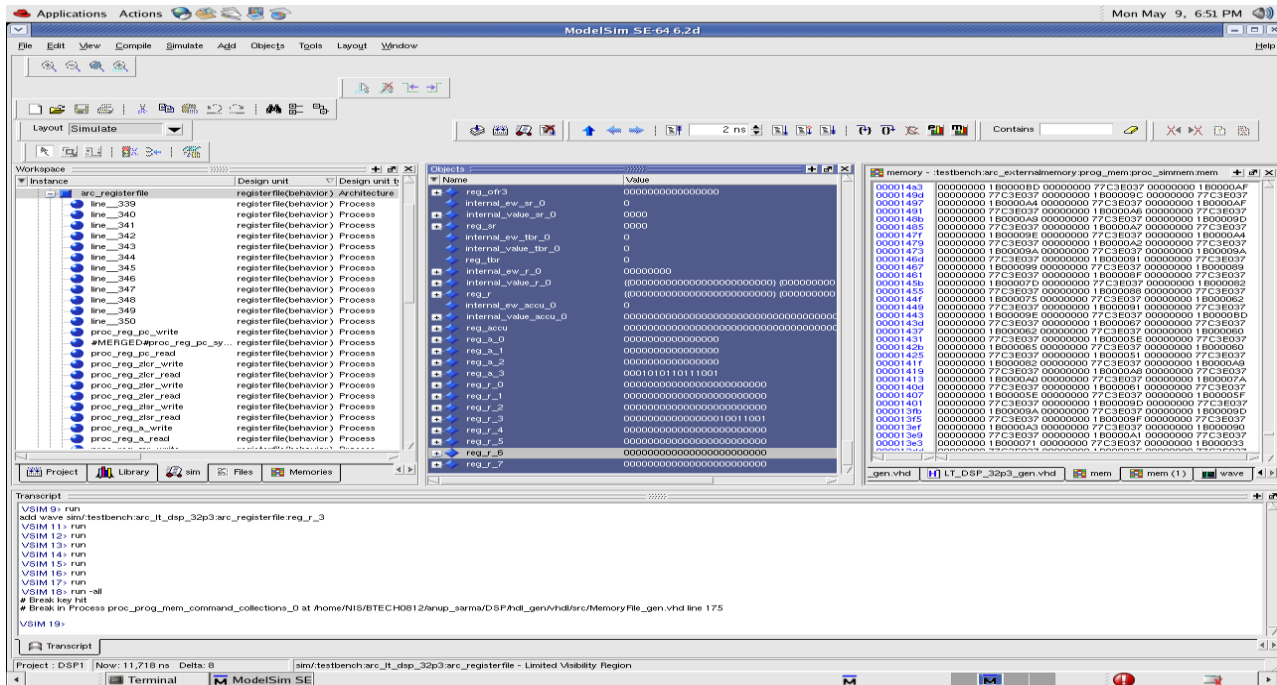
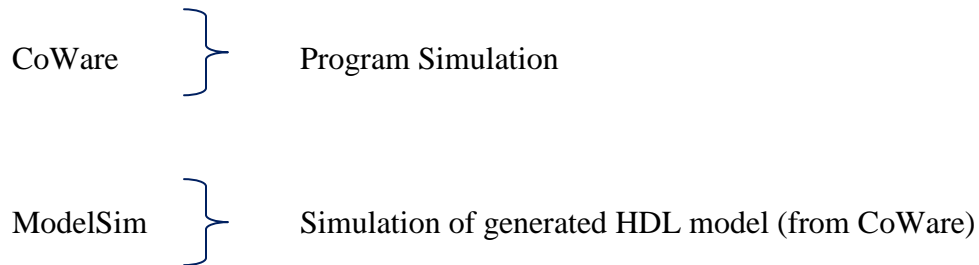


Fig. 9.4 ModelSim Screenshot

Chapter 10

RESULTS

This chapter shows the simulation results carried out on the following platforms:



Additionally Matlab is used for reading of input image as well as display of processed output image

The *first* step was running the program on CoWare Processor Debugger as described in the previous chapters. The result obtained was in the form of the new pixel values stored in the data memory of the simulated processor.

The *second* step was simulation of generated HDL model. The HDL files for the simulated processor were automatically generated by CoWare Processor Generator. These files along with memory configuration files (for program memory and data memory) were used to simulate the processor model in ModelSim.

The *third* step was generation of the processed image in Matlab and comparison with Matlab processed image using in-built functions. The new pixel values stored in the data memory after simulation in ModelSim was suitably inputted into Matlab for the image generation. Finally, both the Matlab processed image and Processor Output image were compared in view of their respective histograms and the error/difference image.

Image 1

Image size: 64 x 64

Total number of pixels: 4096

Type: Greyscale

Pixel Intensity range: 0-255

Total number of Intensity Levels: 256



**Note: An enlarged view of the 64x64 images is shown above*

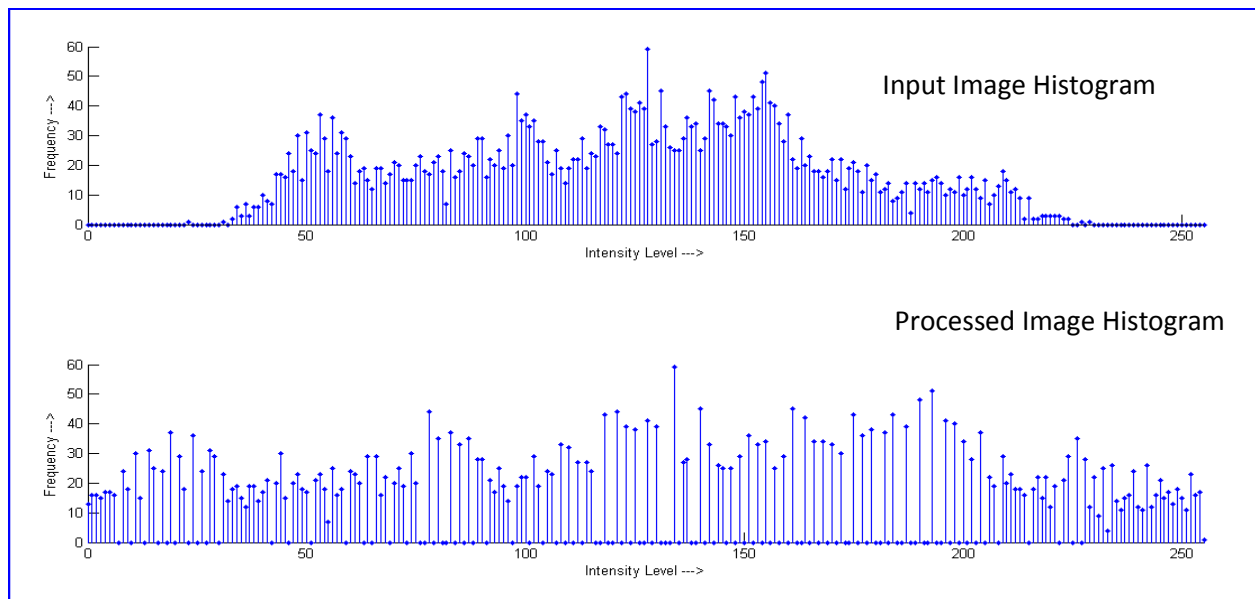


Fig.10.5Histogram Comparison of 64x64 image

Image 2

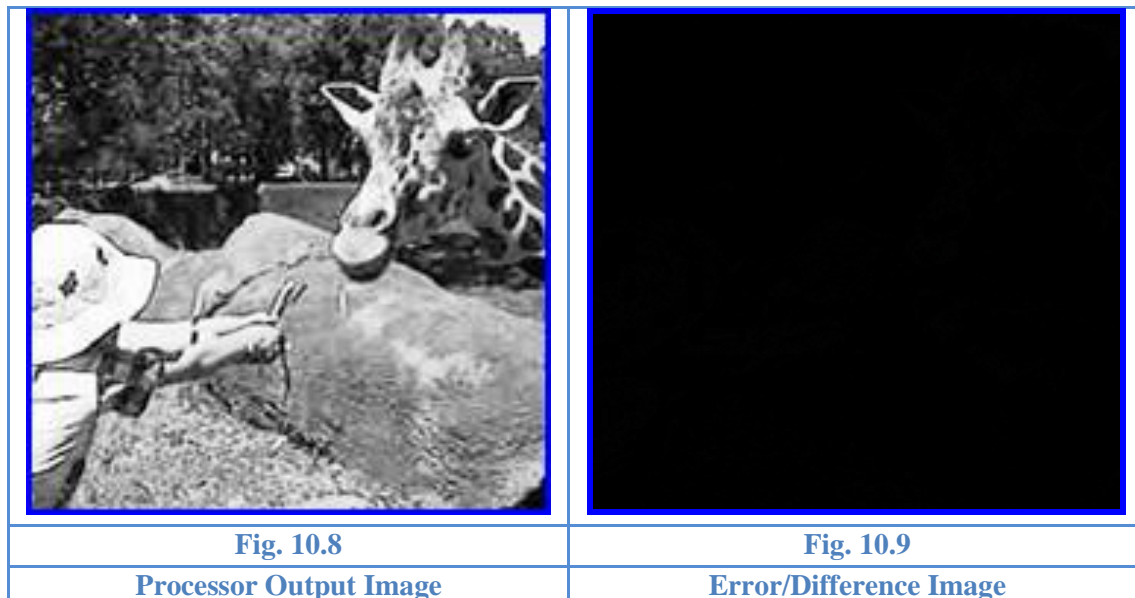
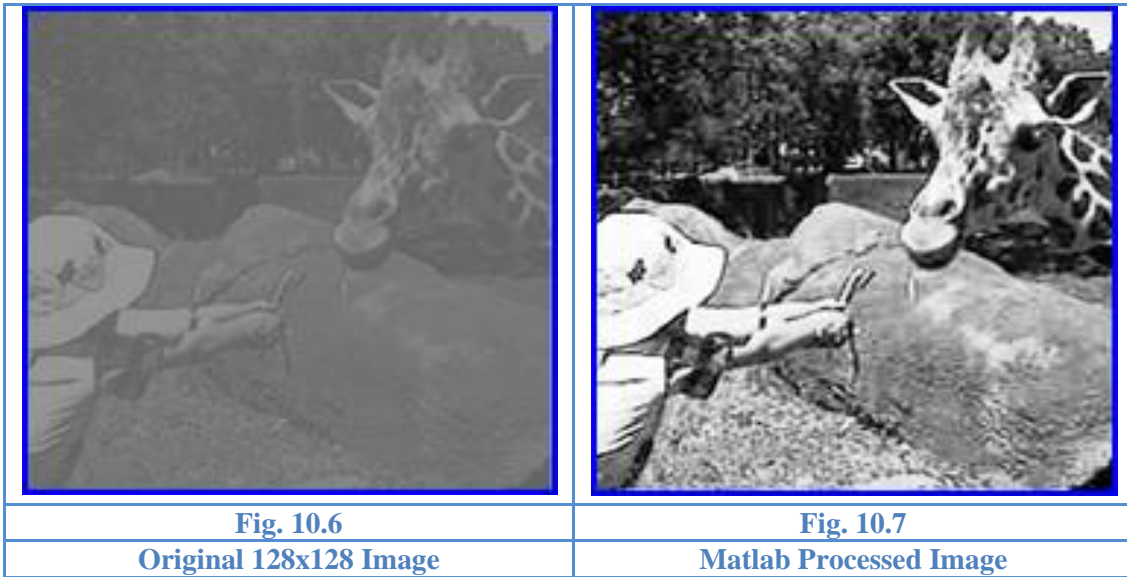
Image size: 128 x 128

Total number of pixels: 16384

Type: Greyscale

Pixel Intensity range: 0-255

Total number of Intensity Levels: 256



*Note: An enlarged view of the 128x128 images is shown above

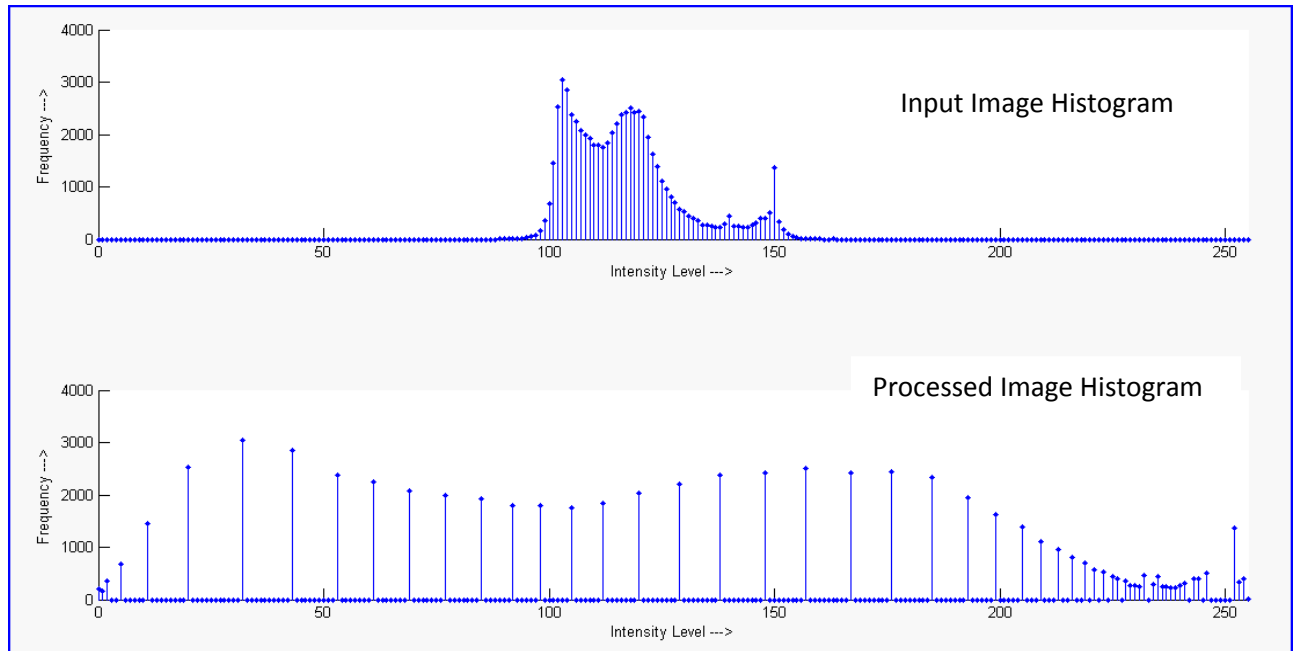


Fig.10.10 Histogram Comparison of 128x128 image

Thus, comparison of histogram equalized output from Matlab and the one obtained by running the same algorithm through our processor designed is shown. The difference between the two images has been shown as the error /difference image. The error image containing almost all dark regions imply that algorithm has been implemented successfully. However few light spots in the difference image can be explained in terms of the fact that Matlab uses a high degree of precision while handling fractional numbers (associated with the division process) while our design incorporates division simply by shift right operation hence fractional part is not taken care of resulting in a loss of precision.

Also the image histogram of input and output images are shown separately. In case of image 1(lena 64x64) it is seen that pixel values are spread in the region 50-200 range. While histogram equalized image shows a more uniform spread of intensity values occupying 0-255 range.

In case of Image 2, input histogram has a very narrow range of intensity values from 100-150, implying that image is a very low contrast one. Corresponding output histogram has pixel values spreading the entire 0-255 range. This is also verified by the apparent improvement in subjective appearance of the output image as can be seen from the figure.

Chapter 11

SYNTHESIS AND PHYSICAL IMPLEMENTATION

In order to verify the functionality of the processor at real time, the processor had to be implemented on hardware. The hardware implementation is done by running the processor on an FPGA board along with a sample program that is sufficient to verify the instruction set of the processor. The status of the running processor is verified by using ChipScope Pro.

To do so, we need to understand the HDL structure of the generated model. The following diagram gives an overview of the generated architecture.

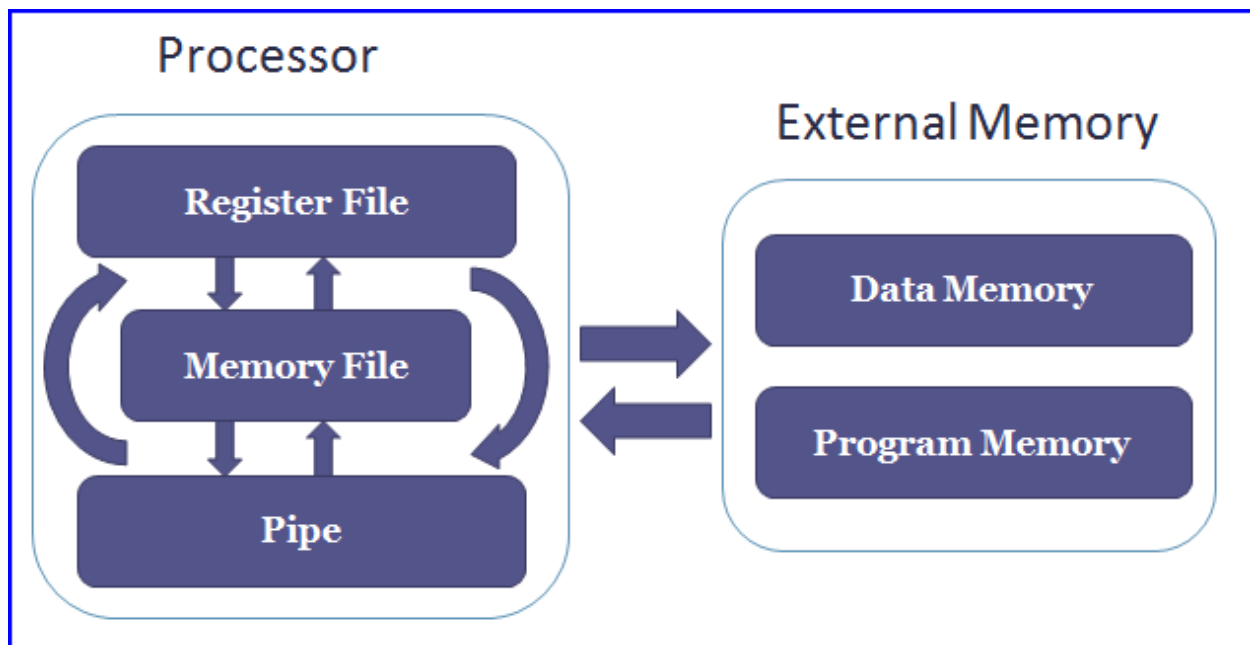


Fig. 11.1 VHDL File structure

As shown in figure, the VHDL model consists of two parts – the processor containing the Register File, Memory File and the Pipe File as component which has a communication mechanism among themselves. The External Memory consists of Data and Program memory which communicates with processor through the memory file. However, the memories generated by Processor Generator are meant for simulation purpose only and need to be replaced by hardware specific memory models during implementation phase.

Thus, we use Xilinx IP core generator to generate Block RAM memory models for both data and program memory. This generates memory files according to the specified width and depth of memory required. Although it is possible to write the memory files by ourselves, the advantage of using IP core is that memory initialisation can be done through ‘.coe’ file during core generation process.

Now a project is created in Xilinx ISE where the top module consists of processor model and the BRAM memories generated as component. Appropriate port mapping has to be performed while replacing the simulation memories with synthesizable memories. The following results are obtained during the process of synthesis. The target device is Virtex 2 Pro (XCV2P30).

Logic Usage (Number)	Used	Available	Utilization(%)
Slices	2445	13696	18
Slice flip flops	684	27392	2
4 input LUTs	5490	27392	20
Bonded IOB	6	556	1
BRAMs	14	136	10
MULT18X18s	4	136	2
GCLKs	2	16	12
DCMs	1	8	12

Table11.1 Device Utilisation Summary

11.1 RTL Schematic

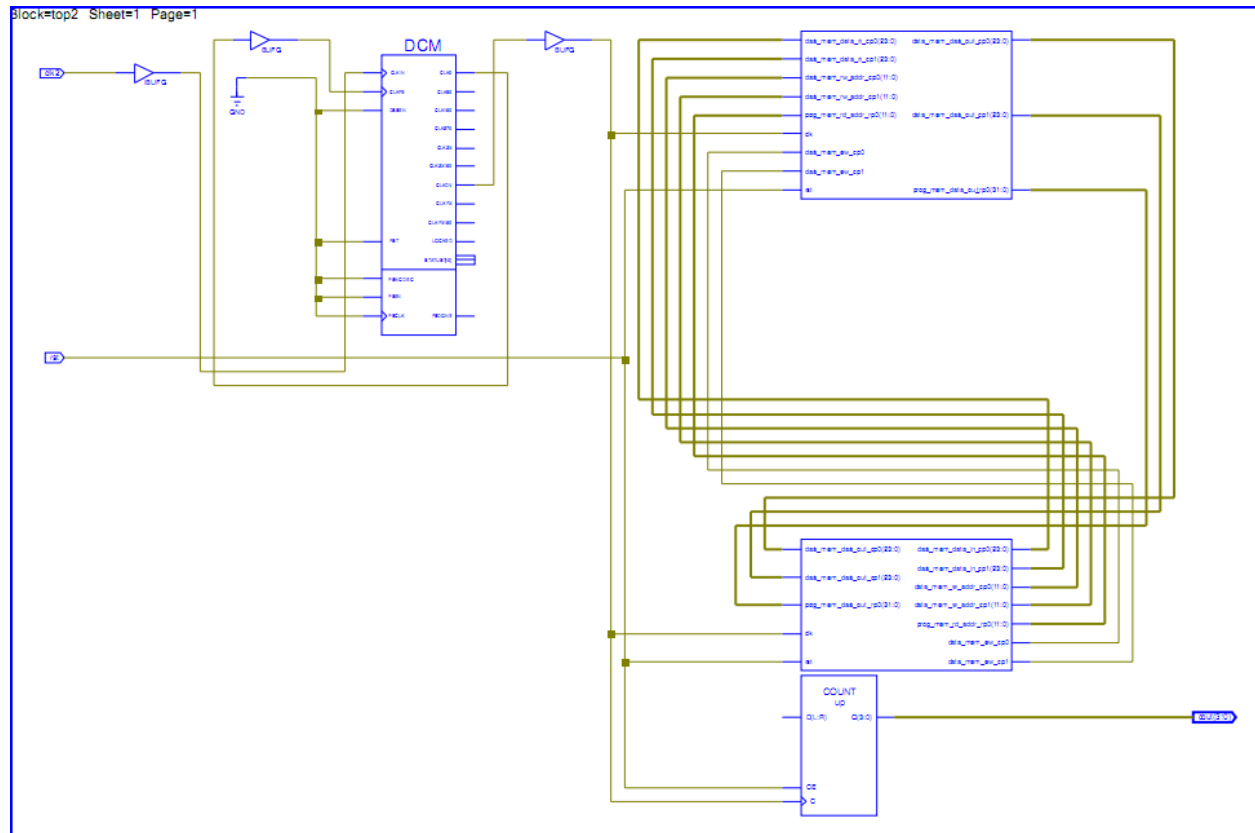


Fig.11.2 Complete RTL Schematic

As seen from the RTL schematic, there are four different blocks. Two obvious blocks are the processor and the memory block. The other two correspond to DCM (digital clock manager) and a dummy counter.

Before we discuss the importance of the DCM and dummy counter we look into RTL of processor and memory block.

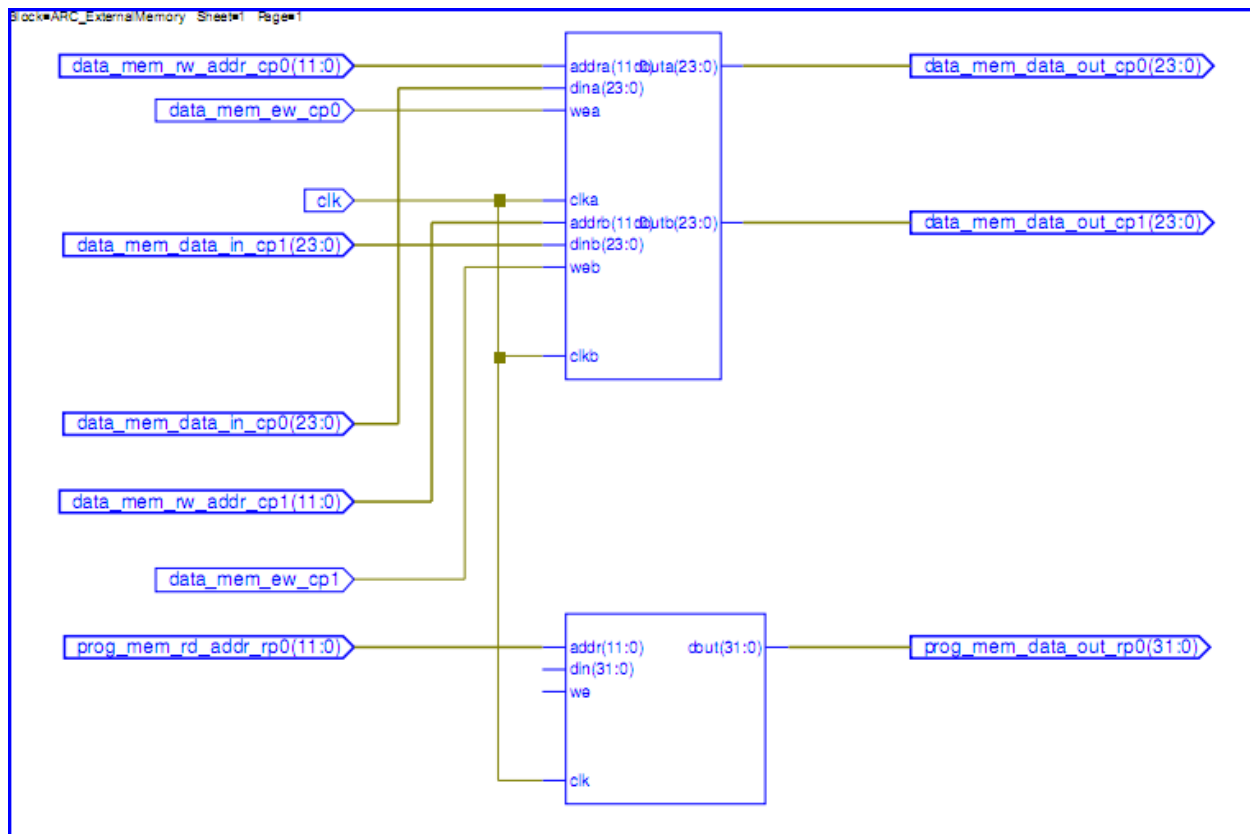


Fig. 11.3 RTL Schematic of Memory block

Here program memory is defined as single port BRAM of width 32 and depth 4096, so that it has 12 address lines. The program memory is essentially behaves like a ROM, hence data in and write enable pins are left unconnected.

The program memory is defined to be a dual port Block RAM of width 32, depth 4096 .It has two independent clocks 'clka' and 'clkb', but connected to same global clock 'clk'.

The RTL schematic of the processor is shown in the next page.

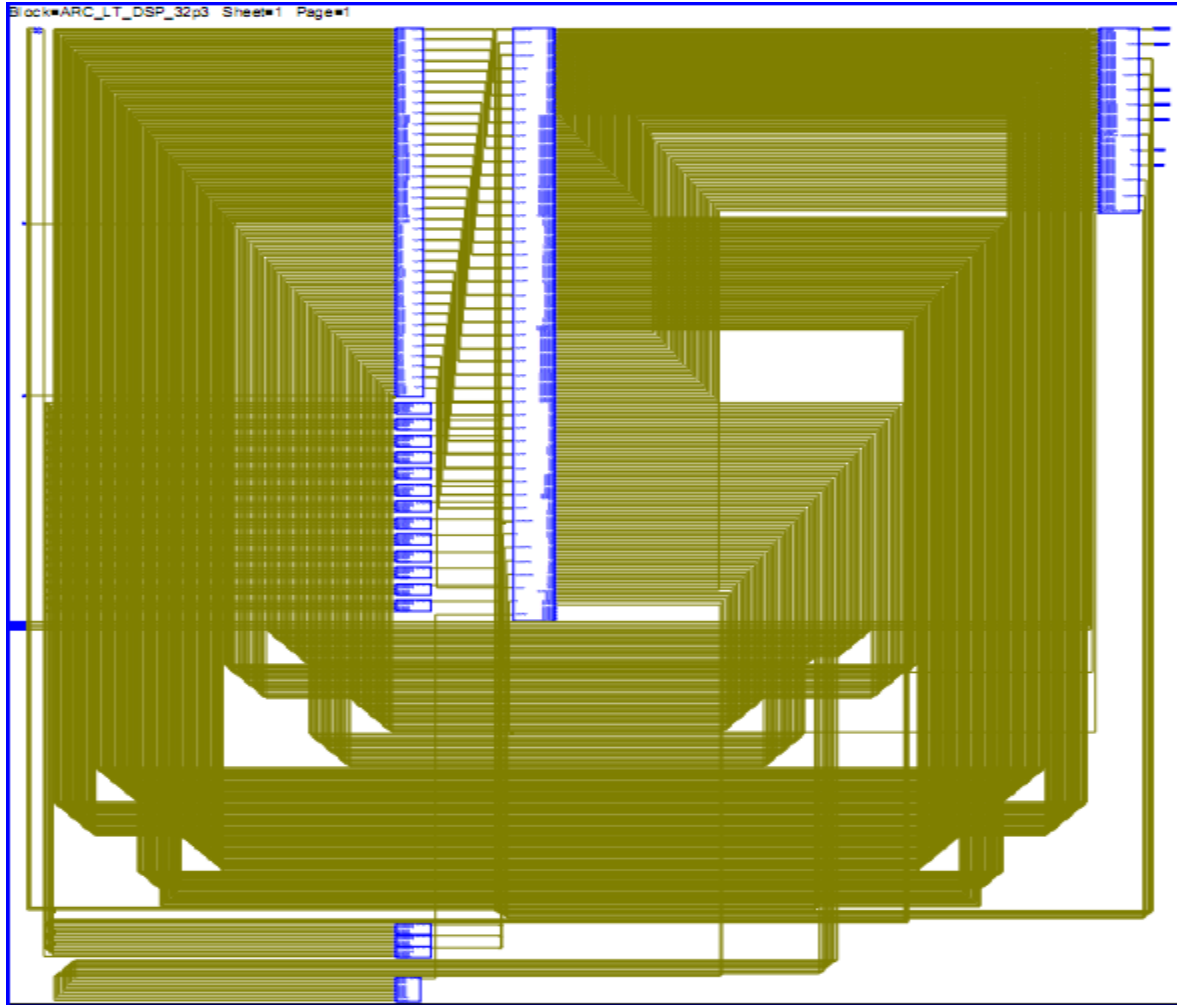


Fig. 11.4 Processor RTL Schematic

The Digital clock manager(DCM) primitive in Xilinx FPGA parts capable of implementing clocked DLL (delay locked loop), a digital frequency synthesizer, digital phase shifter or a digital spread spectrum. The necessity of DCM comes from the fact that our design can work at a maximum frequency of 82.07 MHz whereas the onboard clock frequency of Virtex 2 pro FPGA board is 100MHz. Thus we have to use a frequency divider that can divide the onboard frequency to a value less than 100MHz. Here, we specify a division parameter of 4 such that processor actually runs at 25MHz.

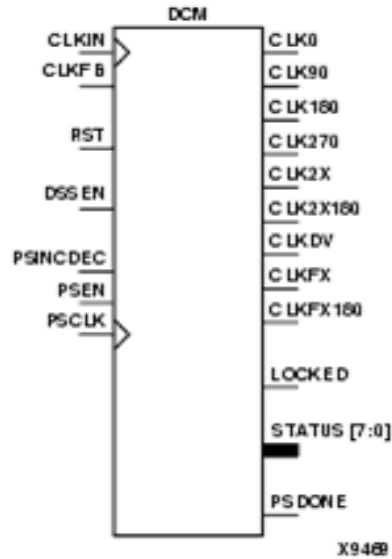


Fig. 11.5 Digital Clock Manager

We have also a dummy counter running .This is in order to provide a output port to our top design which otherwise doesn't synthesize. The other ports to our top design include 'rst' and 'clk' which act as input port.

Once completed, the entire design is downloaded on to the FPGA board. The output is verified using Chip Scope pro which is found to be same as expected.

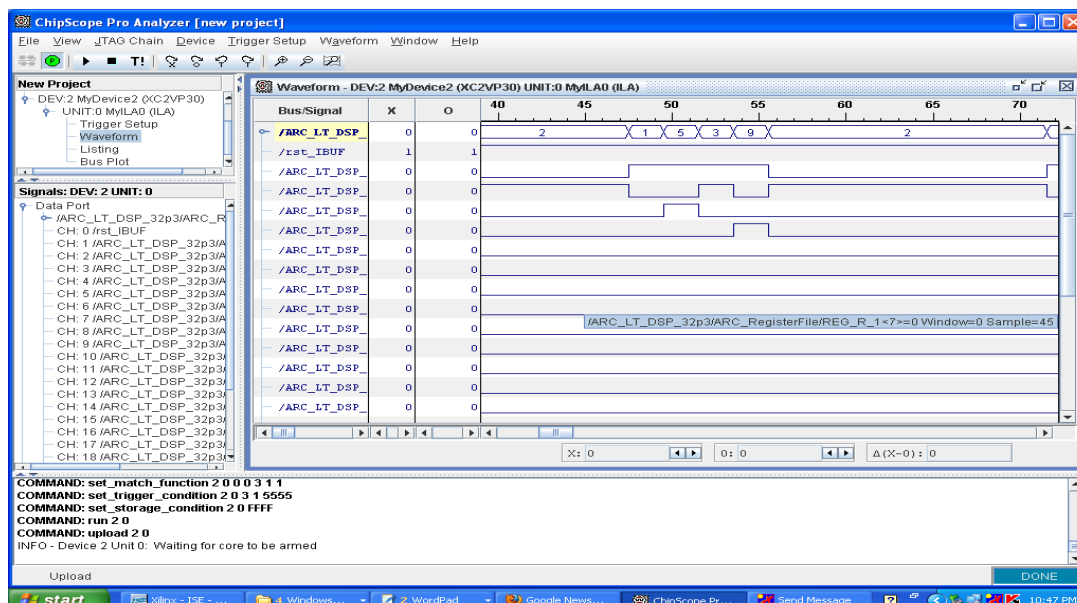


Fig. 11.6ChipScope Pro Screenshot

Chapter 12

CONCLUSION AND FUTURE WORK

The project was based on designing an Application Specific Instruction-SetProcessor whose instructions were tailored made for a specific application. We took Histogram Equalization in image processing as a sample application and showed operation of the designed processor on various image sizes with its specific instruction set. The simulation was carried out on different platforms which included CoWare Processor Debugger and Mentor Graphics ModelSim. This verified the compatibility between the LISA code used to describe the architecture and generated HDL structure. Following the simulation work, the entire model was synthesized using Xilinx ISE and dumped on to a FPGA board for functional verification, which was carried out successfully.

There are a number of ways in which the scope of the current project can be extended. One such possible extension includes processing of colour images. Our current system incorporates a single processor operating on 8 bit greyscale image. It is possible to have three such processors running in parallel and operating on three individual components R, G, B to produce the desired result.

Another improvement that can be made over this system is to design a real time I/O system. Such a system would allow sending and receiving images at real time. Incorporating further image processing algorithms such as object detection and object recognition, could allow usage of such a system in a real time surveillance system.

REFERENCES

- [1] A. Hoffman, F. Friedler, A. Nohl and Surender Parupalli. A Methodology and Tooling Enabling Application Specific Processor Design. In *Proc. of the VLSID*, 2005
- [2] Li Zhang, Shuangfei Li, Zan Yin and Wenyuan Zhao. A Research on an ASIP Processing Element Architecture Suitable for FPGA Implementation. In *Proc. of the International Conference on Computer Science and Software Engineering*, 2008.
- [3] LISA Language Reference Manual supplied by CoWare, Inc
- [4] CoWare, Inc. <http://www.coware.com>
- [5] Xilinx, Inc. <http://www.xilinx.com>
- [6] User Reference Manual, Virtex 2 Pro supplied by Xilinx, Inc
- [7] User Reference Manual, Digital Clock Manager by Xilinx Inc
- [8] ChipScope Pro 12.1 User Software and Cores User Guide by Xilinx Inc
- [9] Wikipedia. <http://en.wikipedia.org>